

Formal Approaches to Ensuring the Safety of Space Software

Ewen Denney & Bernd Fischer
{edenney|fisch}@email.arc.nasa.gov
Robust Software Engineering Group
USRA/RIACS, NASA Ames Research Center



Joint work with I. Schumann, M. Whalen, and many others

A peek into the future...

NEW YORK, WEDNESDAY, SEPTEMBER 18, 2003

NASA FINDS FIRST CLUES IN MARS LANDING DISASTER; AUTOMATIC CODE GENERATOR FAILED, SOURCES SAY

NASA Dismissed Researchers Who Investigated its Safety

By David Allen

WASHINGTON, Sept. 17 — NASA officials on Tuesday said they had found the first clues to a mysterious Mars landing disaster, blaming the failure on a software generator that produced faulty code. The generator, which was used to create the software that controlled the Mars rover, was found to have produced a critical error that caused the rover to crash.

The error, which occurred during the final descent of the rover, was caused by a flaw in the code that controlled the rover's airbags. The code, which was generated by a software tool, failed to properly inflate the airbags, causing the rover to crash.

The error was discovered by a team of researchers who had been investigating the disaster. The researchers, who had been dismissed by NASA, had found evidence that the code generator was the cause of the problem.

The researchers had found that the code generator had produced a critical error that caused the rover to crash. The error was discovered by a team of researchers who had been investigating the disaster.

The researchers had found that the code generator was the cause of the problem. The error was discovered by a team of researchers who had been investigating the disaster.

Manned Mission On Hold for Now

By Judith A. Smit

WASHINGTON, Sept. 17 — NASA officials on Tuesday said they had found the first clues to a mysterious Mars landing disaster, blaming the failure on a software generator that produced faulty code. The generator, which was used to create the software that controlled the Mars rover, was found to have produced a critical error that caused the rover to crash.

The error, which occurred during the final descent of the rover, was caused by a flaw in the code that controlled the rover's airbags. The code, which was generated by a software tool, failed to properly inflate the airbags, causing the rover to crash.

The error was discovered by a team of researchers who had been investigating the disaster. The researchers, who had been dismissed by NASA, had found evidence that the code generator was the cause of the problem.

The researchers had found that the code generator had produced a critical error that caused the rover to crash. The error was discovered by a team of researchers who had been investigating the disaster.

The researchers had found that the code generator was the cause of the problem. The error was discovered by a team of researchers who had been investigating the disaster.

A peek into the future...

NEW YORK, WEDNESDAY, SEPTEMBER 18, 2003

NASA FINDS FIRST CLUES IN MARS LANDING DISASTER; AUTOMATIC CODE GENERATOR FAILED, SOURCES SAY

NASA Dismissed Researchers Who Investigated its Safety

By David Allen

WASHINGTON, Sept. 17 — NASA officials on Tuesday said they had found the first clues to a mysterious Mars landing disaster, blaming the failure on a software generator that produced faulty code. The generator, which was used to create the software that controlled the Mars rover, was found to have produced a critical error that caused the rover to crash.

The error, which occurred during the final descent of the rover, was caused by a flaw in the code that controlled the rover's airbags. The code, which was generated by a software tool, failed to properly inflate the airbags, causing the rover to crash.

The error was discovered by a team of researchers who had been investigating the disaster. The researchers, who had been dismissed by NASA, had found evidence that the code generator was the cause of the problem.

The researchers had found that the code generator had produced a critical error that caused the rover to crash. The error was discovered by a team of researchers who had been investigating the disaster.

The researchers had found that the code generator was the cause of the problem. The error was discovered by a team of researchers who had been investigating the disaster.

Manned Mission On Hold for Now

By Judith A. Smit

WASHINGTON, Sept. 17 — NASA officials on Tuesday said they had found the first clues to a mysterious Mars landing disaster, blaming the failure on a software generator that produced faulty code. The generator, which was used to create the software that controlled the Mars rover, was found to have produced a critical error that caused the rover to crash.

The error, which occurred during the final descent of the rover, was caused by a flaw in the code that controlled the rover's airbags. The code, which was generated by a software tool, failed to properly inflate the airbags, causing the rover to crash.

The error was discovered by a team of researchers who had been investigating the disaster. The researchers, who had been dismissed by NASA, had found evidence that the code generator was the cause of the problem.

The researchers had found that the code generator had produced a critical error that caused the rover to crash. The error was discovered by a team of researchers who had been investigating the disaster.

The researchers had found that the code generator was the cause of the problem. The error was discovered by a team of researchers who had been investigating the disaster.

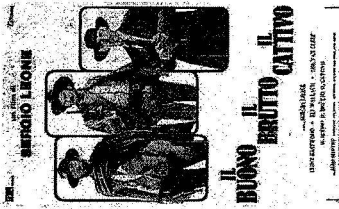
Outline

1. Introduction
(or: Taking Stock)
2. Certifiable Program Generation
(or: I have a plan)
3. Certification Framework
(or: Greek Letters)
4. Annotation Generation
(or: TANSTAAFL)
5. Experiments
(or: Drosophila and Tables)
6. Future Work
(or: Wild Speculations)

Taking Stock: The Good, the Bad, the Ugly

The Good: It hasn't happened yet!

- no accidents caused by generated code



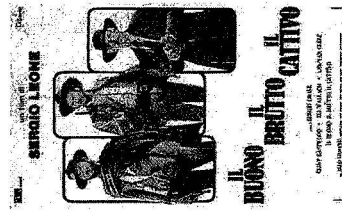
Taking Stock: The Good, the Bad, the Ugly

The Good: It hasn't happened yet!

- no accidents caused by generated code

The Bad: It hasn't happened yet!

- limited generator capabilities: glorified pretty-printers
- limited generator usage
- excessive post-hoc validation



The Ugly: It will happen!

- too many bug reports (cf. optimizing compilers):

Notice the function "beforeStart" should return a boolean but doesn't. Since this code was generated by Netbeans it's not editable...

- too many generators (www.codegeneration.net: ~ 200)
- increasing application pressure: model-driven architecture

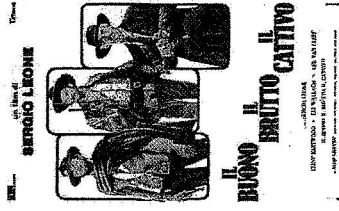
Taking Stock: The Good, the Bad, the Ugly

The Good: It hasn't happened yet!

- no accidents caused by generated code

The Bad: It hasn't happened yet!

- limited generator capabilities: glorified pretty-printers
- limited generator usage
- excessive post-hoc validation



Taking Stock: The Correctness Dilemma

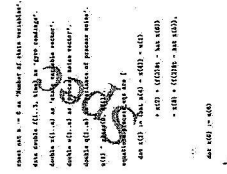
Do you trust your code generator?

- Correctness of generated code depends on correctness of generator
- Correctness of generator difficult to show practically

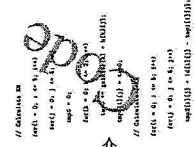
– very large

– very complicated

– very dynamic



to what to do?



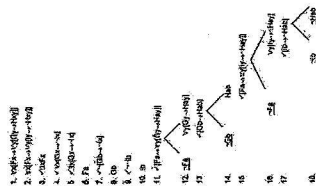
Generator Assurance Approaches (I)

Correctness-by-Construction:

Generator is based on logical framework; code is derived by correctness-preserving transformations

Techniques:

- deductive program synthesis
- refinement and transformation systems
- translation verification



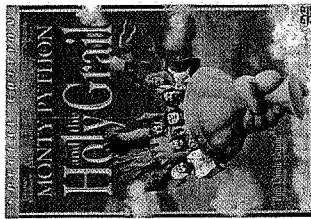
Generator Assurance Approaches (I)

Correctness-by-Construction:

Generator is based on logical framework; code is derived by correctness-preserving transformations

Techniques:

- deductive program synthesis
- refinement and transformation systems
- translation verification



Advantages:

- highest degree of confidence ("proofs-as-programs")

Disadvantages:

- expensive — systems difficult to build & maintain
- opaque — correctness argument convoluted and buried in generator (\Rightarrow must trust generator)

Generator Assurance Approaches (II)

Generator Qualification:

Generator is tested to same level of orthodoxy as generated code

Advantages:

- currently only approach accepted by FAA
- currently state-of-practice

Disadvantages:

- expensive — testing efforts very high
- expensive — re-qualification required after changes
- limited — only partial assurance
- opaque — no explicit correctness argument (\Rightarrow must trust generator)

Taking Stock: The Correctness Dilemma (revisited)

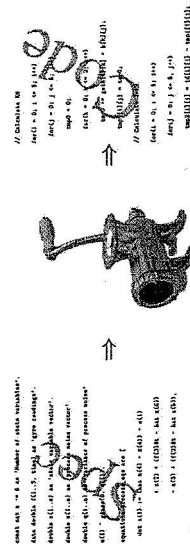
Do you trust your code generator?

- Correctness of generated code depends on correctness of generator
- Correctness of generator difficult to show practically

— very large

— very complicated

— very dynamic



So what?

- Don't care whether generator is buggy for other people as long as it works for me now!

\Rightarrow Certifiable Program Generation

Certifiable Program Generation

basic Idea I:

Certify generated programs individually, not the generator

⇒ product-oriented approach rather than process-oriented

⇒ no need to re-certify generator

⇒ minimizes trusted component base

Certifiable Program Generation

basic Idea I:

Certify generated programs individually, not the generator

basic Idea II:

Extend the generator to support certification

⇒ generate code with additional “mark-up”

⇒ CAVEAT: keep certification independent from code generation

Certifiable Program Generation

basic Idea I:

Certify generated programs individually, not the generator

basic Idea II:

Extend the generator to support certification

basic Idea III:

Use Floyd-Hoare program verification techniques

⇒ rigorous mathematical foundation

⇒ proofs are independently verifiable evidence (*certificates*)

⇒ code mark-up gives hints only

⇒ code mark-up $\hat{=}$ pre-/post-conditions, loop invariants

Certifiable Program Generation

basic Idea I:

Certify generated programs individually, not the generator

basic Idea II:

Extend the generator to support certification

basic Idea III:

Use Floyd-Hoare program verification techniques

basic Idea IV:

Focus on specific *safety* properties

- array bounds, partial operators, ...
 - variable initialization, def-use, ...
 - physical units, frames, ...
 - volatile memory restrictions, ...
 - vector norms, matrix symmetry, ...
 - ...
- } language-specific
 } domain-specific

Generator Assurance Approaches (III)

Certifiable Program Generation:

Generator is extended to generate code with extra artefacts that support an independent assurance demonstration

related techniques:

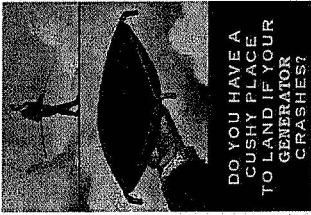
- result checking
- proof-carrying code

Advantages:

- customizable — different safety properties
- transparent — explicit safety arguments
- high degree of assurance — formal proofs

Disadvantages:

- limited — only partial assurance (flip-side of customizable)

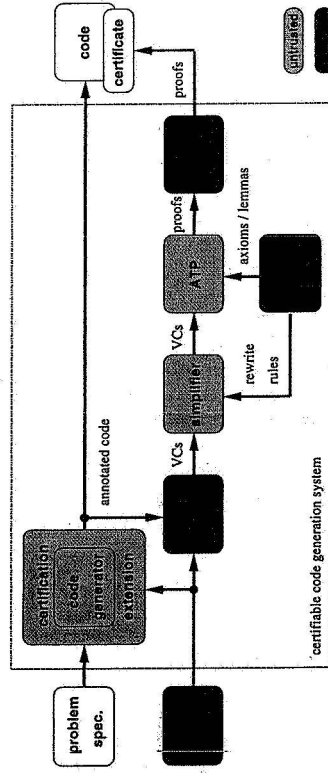


Generator Assurance Architectures



Correct-by-construction: "Trust me, I'm a doctor..."

Generator Assurance Architectures



Certifiable program generation: "Don't trust me, I'm a computer scientist..."

- *Trusted code base* minimized
 - "large" components untrusted
 - trusted components (more) deterministic
- Approach
 - generate safety obligations (i.e., VCG applies safety policy to program)
 - simplify, prove, & check

Outline

1. Introduction
(or: Taking Stock)
2. Certifiable Program Generation
(or: I have a plan)
3. Certification Framework
(or: Greek Letters)
4. Annotation Generation
(or: TASTAAFL)
5. Experiments
(or: Drosophila and Tables)
6. Future Work
(or: Wild Speculations)

Certification Framework

safety property: operational characterization of *intuitively safe* programs

“All automatic variables shall have been assigned a value before being used”
(MISRA 9.1)

normal:

- introduce “shadow variables” to record safety information
- operational semantics (extended by effects on shadow variables):

$$\begin{aligned} \langle x := e, \eta, \bar{\eta} \rangle &\Rightarrow \langle \text{skip}, \eta \oplus \{x \mapsto [e]_{\eta}\}, \bar{\eta} \oplus \{x_{\text{init}} \mapsto \text{INIT}\} \rangle \\ \langle x[e_1] := e_2, \eta, \bar{\eta} \rangle &\Rightarrow \langle \text{skip}, \eta \oplus \{x \mapsto (x \oplus \{[e_1]_{\eta} \mapsto [e_2]_{\eta}\})\}, \\ &\quad \bar{\eta} \oplus \{x_{\text{init}} \mapsto (x_{\text{init}} \oplus \{[e_1]_{\eta} \mapsto \text{INIT}\})\} \rangle \end{aligned}$$

...

Certification Framework

safety property: operational characterization of *intuitively safe* programs

“All automatic variables shall have been assigned a value before being used”
(MISRA 9.1)

normal:

- introduce “shadow variables” to record safety information
- operational semantics (extended by effects on shadow variables)
- semantic safety definition (judgement on expressions and statements)
- safety reduction (consistency of safety property):

$$\eta, \bar{\eta} \models c \text{ safe and } \langle c, \eta, \bar{\eta} \rangle \Rightarrow \langle c', \eta', \bar{\eta}' \rangle \text{ implies } \eta', \bar{\eta}' \models c' \text{ safe}$$

\Rightarrow “safe programs don’t go wrong”

Certification Framework

safety property: operational characterization of *intuitively safe* programs

“All automatic variables shall have been assigned a value before being used”
(MISRA 9.1)

normal:

- introduce “shadow variables” to record safety information
- operational semantics (extended by effects on shadow variables)
- semantic safety definition (judgement on expressions and statements):

$$\begin{aligned} \eta, \bar{\eta} \models x \text{ safe}_{\text{init}} &\quad \text{iff } x_{\text{init}} = \text{INIT} \\ \eta, \bar{\eta} \models x[e] \text{ safe}_{\text{init}} &\quad \text{iff } \bar{\eta}(x_{\text{init}})[e]_{\eta, \bar{\eta}} = \text{INIT and } \eta, \bar{\eta} \models e \text{ safe}_{\text{init}} \end{aligned}$$

...

$$\eta, \bar{\eta} \models x[e_1] := e_2 \text{ safe}_{\text{init}} \quad \text{iff } \eta, \bar{\eta} \models e_1 \text{ safe}_{\text{init}} \text{ and } \eta, \bar{\eta} \models e_2 \text{ safe}_{\text{init}}$$

...

Certification Framework

safety policy: proof rules to show that safety property holds for program

- responsible for
 - maintenance of shadow variables
 - construction of safety obligations
- Hoare-rules (extended by safety predicate and shadow variables):

$$\begin{aligned} (\text{assign}) \quad & \frac{Q[e/x, \text{INIT}/x_{\text{init}}] \wedge \text{safe}_{\text{init}}(e) \{x := e\} Q}{Q} \\ (\text{update}) \quad & \frac{Q \left[\frac{\text{upd}(x, e_1, e_2)/x, \wedge \text{safe}_{\text{init}}(e_1) \wedge \text{safe}_{\text{init}}(e_2) \{x[e_1] := e_2\} Q}{\text{upd}(x_{\text{init}}, e_1, \text{INIT})/x_{\text{init}}} \right]}{Q} \\ (\text{if}) \quad & \frac{P \Rightarrow \text{safe}_{\text{init}}(b) \quad b \wedge P \{c\} Q \quad \neg b \wedge P \Rightarrow Q}{P \{ \text{if } b \text{ then } c \} Q} \\ (\text{while}) \quad & \frac{P \Rightarrow \text{safe}_{\text{init}}(b) \quad b \wedge P \{c\} P}{P \{ \text{while } b \text{ do } c \} \neg b \wedge P} \end{aligned}$$

Certification Framework

safety policy: proof rules to show that safety property holds for program

- responsible for
 - maintenance of shadow variables
 - construction of safety obligations
- Hoare-rules (extended by safety predicate and shadow variables)
- safety predicate $\text{safe}_{\text{init}}(e)$ corresponds to semantic safety conditions:

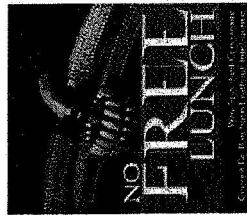
$$\begin{aligned} \text{safe}_{\text{init}}(x) &\equiv x_{\text{init}} = \text{INIT} \\ \text{safe}_{\text{init}}(x[e]) &\equiv x_{\text{init}}[e] = \text{INIT} \wedge \text{safe}_{\text{init}}(e) \end{aligned}$$

...

Annotation Generation

The Certification Dilemma:

Annotations are crucial but cannot be invented by the machinery.



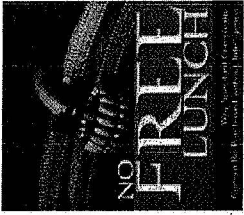
Annotation Generation

The Certification Dilemma:

Annotations are crucial but cannot be invented by the machinery and must (ultimately) be provided by the generator developer.

The Bad: It is hard work!

- annotation generation is tedious meta-hack^HH^HH programming
- annotations are cross-cutting concerns (object- and meta-level)
- annotations are different for each safety property



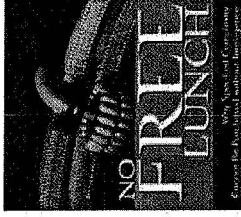
Annotation Generation

The Certification Dilemma:

Annotations are crucial but cannot be invented by the machinery and must (ultimately) be provided by the generator developer.

The Bad: It is hard work!

- annotation generation is tedious meta-hack^HH^HH programming
- annotations are cross-cutting concerns (object- and meta-level)
- annotations are different for each safety property



The Good: Everything is known at meta-compile time!

- structure and purpose of generated code limited and known
- safety properties limited and known

Annotation Generation

Example: annotations for *array-safety*:

```
begin
  var c[C], w[N,C];
  ...
  for i := 1 : N do // pick classes randomly
    c[i] := rnd(C);
    ...
    for j := 1 : N do // set weight for picked class
      for j := 1 : C do w[i,j] := 0.0;
      w[i,c[i]] := 1.0;
      ...
    end
```

Annotation Generation

Example: annotations for *array-safety*:

```
begin
  var c[C], w[N,C];
  ...
  for i := 1 : N do // pick classes randomly
    c[i] := rnd(C);
    ...
    for i := 1 : N do // set weight for picked class
      // inv:  $1 \leq c[i] \leq C$ 
      for j := 1 : C do w[i,j] := 0.0;
      w[i,c[i]] := 1.0;
      ...
    end
```

Annotation Generation

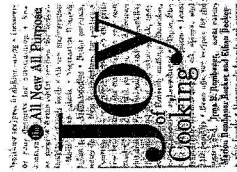
Example: annotations for *array-safety*:

```
begin
  var c[C], w[N,C];
  ...
  for i := 1 : N do // pick classes randomly
    c[i] := rnd(C);
    // post:  $\forall j. 1 \leq j \leq N \Rightarrow 1 \leq c[j] \leq C$ 
    ...
    for i := 1 : N do // set weight for picked class
      // inv:  $1 \leq c[i] \leq C$ 
      for j := 1 : C do w[i,j] := 0.0;
      w[i,c[i]] := 1.0;
      ...
    end
```

Annotation Generation (Meta-level)

Overall recipe:

- Repeat until all generated VCs are proven
1. identify structure and location of required annotations in code
2. for each annotation, generalize it to meta-annotation
3. for each meta-annotation,
 - write annotation template
 - write meta-program that produces annotation
4. for each location, identify the responsible schema(s)
5. for each schema, integrate meta-annotations



Annotation Generation

Example: annotations for *array-safety*:

```
begin
  var c[C], w[N,C];
  ...
  for i := 1 : N do // pick classes randomly
    // inv:  $\forall j. 1 \leq j < i \Rightarrow 1 \leq c[j] \leq C$ 
    c[i] := rnd(C);
    // post:  $\forall j. 1 \leq j \leq N \Rightarrow 1 \leq c[j] \leq C$ 
    ...
    for i := 1 : N do // set weight for picked class
      // inv:  $1 \leq c[i] \leq C$ 
      for j := 1 : C do w[i,j] := 0.0;
      w[i,c[i]] := 1.0;
      ...
    end
```

Annotation Generation (Object-level)

At program generation time:

- annotation templates instantiated in parallel with code templates
 - code generator / frontend
 - annotations refined in parallel with code
 - code generator / backend
 - information propagated “globally” in pre-processing step
 - approximates strongest postcondition transformer
- ⇒ annotations *not* trusted (i.e., not safety-critical)
- obligations produced by (trusted) safety policy

Certification Experiments

Experimental set-up:

- Synthesis systems & test programs:
 - AUTOFILTER: state estimation based on Kalman-filters
 - ds1 – Deep Space 1 attitude estimation
 - iss – Space Station simulation (part)
 - AUTOBAYES: statistical data analysis
 - segm – image segmentation via clustering
 - gauss – image fitting to model
- Safety policies:
 - array: $\forall a[i] \in c \cdot a_{lo} \leq i \leq a_{hi}$
 - init: $\forall read-var \ x \in c \cdot init(x)$
 - inuse: $\forall input-var \ x \in c \cdot use(x)$
 - symm: $\forall matrix-var \ m \in c \cdot \forall i, j. m[i, j] = m[j, i]$
 - norm: $\forall vector-var \ v \in c \cdot \sum_{i=0}^{v_{hi}} v[i] = 1$



Drosophila nigrosparsula

Certification Results

Example	S	P	Policy	A	A*	N	N _{fail}	T _{gen}	T _{proof}
ds1	48	431	array	0	19	1	-	5.5	1
			init	87	444	74	-	11.4	84
			inuse	61	413	21	1	8.1	202
			symm	75	261	865	-	70.8	794
iss	97	755	array	0	19	4	-	24.7	3
			init	88	458	71	-	39.7	88
			inuse	60	361	1	1	31.6	-
			symm	87	274	480	-	66.2	510
segm	17	517	array	0	53	1	-	3.0	1
			init	171	1090	121	-	7.6	109
			norm	195	247	14	-	3.6	12
gauss	18	1039	array	20	505	20	-	21.3	16
			init	118	1615	316	-	54.3	259

Certification Results

Example	S	P	Policy	A	A*	N	N _{fail}	T _{gen}	T _{proof}
ds1	48	431	array	0	19	1	-	5.5	1
			init	87	444	74	-	11.4	84
			inuse	61	413	21	1	8.1	202
			symm	75	261	865	-	70.8	794
iss	97	755	array	0	19	4	-	24.7	3
			init	88	458	71	-	39.7	88
			inuse	60	361	1	1	31.6	-
			symm	87	274	480	-	66.2	510
segm	17	517	array	0	53	1	-	3.0	1
			init	171	1090	121	-	7.6	109
			norm	195	247	14	-	3.6	12
gauss	18	1039	array	20	505	20	-	21.3	16
			init	118	1615	316	-	54.3	259

⇒ formulation of *inuse*-policy too conservative

Certification Results

real errors caught in generator (anecdotal evidence only...):

- division-by-zero error hidden in schema:
 - generated fragment:


```
for i := 1 : C do
  c[i] := x[rnd(N)];
for i := 1 : N do
  for j := 1 : C do
    w[i,j] := sqrt((c[j]-x[i])**2)
    / sum(k := 1 : C, sqrt((c[k]-x[i])**2));
```
 - ⇒ error manifests itself only if all input data $x[i]$ are equal
 - ⇒ caught by *partial-operator*-policy
- uninitialized variable caused by generator maintenance:
 - added simplified version of Kalman-schema (hardcodes $H = 0$)
 - botched ‘partial evaluation’: removed too much code
 - ⇒ caught by *init*-policy right after introduction

Future Directions

- Extend range or safety policies
 - type conformance: units, behavioral subtypes, ...
 - protocol conformance: locking, separation, ...
- Support different “reasoning engines”: static analysis
- Apply to other code generators: Simulink/Matlab RealTime Workshop
- Annotation inference
 - separate code generation and annotation generation
 - infer annotations from code structure and safety policy
 - use AOP-style techniques
 - “go meta-meta”: generate aspects if necessary

⇒ exploits idiomatic structure of generated code

Future Directions

- Extend range or safety policies
 - type conformance: units, behavioral subtypes, ...
 - protocol conformance: locking, separation, ...
- Support different “reasoning engines”: static analysis
- Apply to other code generators: Simulink/Matlab RealTime Workshop
- Annotation inference
 - separate code generation and annotation generation
 - infer annotations from code structure and safety policy
 - use AOP-style techniques
 - “go meta-meta”: generate aspects if necessary

⇒ exploits idiomatic structure of generated code

PCC for code generators!